Batavia Documentation

Release 3.4.0-dev.19

Russell Keith-Magee

May 31, 2020

Contents

Batavia is an early alpha project. If it breaks, you get to keep all the shiny pieces.

Batavia is an implementation of the Python virtual machine, written in JavaScript. It enables you to run Python bytecode in the browser.

It honors Python 3.4 syntax and conventions, but also provides the ability to reference objects and classes defined natively in JavaScript.

CHAPTER 1

Table of contents

1.1 Tutorial

Get started with a hands-on introduction for beginners

1.2 How-to guides

Guides and recipes for common problems and tasks, including how to contribute

1.3 Background

Explanation and discussion of key topics and concepts

1.4 Reference

Technical reference - commands, modules, classes, methods

CHAPTER 2

Community

Batavia is part of the BeeWare suite. You can talk to the community through:

- @pybeeware on Twitter
- beeware/general on Gitter

2.1 Tutorials

The idea behind Batavia is to take Python bytecode, and ship it to a browser where it runs in a JavaScript implementation of the Python virtual machine.

This means it is possible to write web applications 100% in Python - so those new to web development don't need to learn multiple languages to develop web apps.

These tutorials are step-by step guides for using Batavia in your own projects.

2.1.1 Tutorial: Setting up your Environment

Getting a working local copy of Batavia requires a few steps: getting a copy of the Batavia code, and the ouroboros dependency within a virtual environment.

You'll need to have Python 3.5, 3.6 or 3.7 available for Batavia to work. Instructions on how to set this up are on our Environment setup guide.

1. Setup a *beeware* folder to store everything:

```
$ mkdir beeware
$ cd beeware
```

2. Get a copy of the Batavia code by running a `git clone`:

```
$ git clone https://github.com/beeware/batavia
```

3. We'll need to create a virtual environment, and install Batavia into it.

Linux

```
$ python3.5 -m venv venv
$ . venv/bin/activate
(venv) $ cd batavia
(venv) $ pip install -e .
```

MacOS

```
$ python3.5 -m venv venv
$ . venv/bin/activate
(venv) $ cd batavia
(venv) $ pip install -e .
```

Windows

> py -3.5 -m venv venv > venvScriptsactivate (venv) > cd batavia (venv) > pip install -e .

4. In addition, you need to install Node.js. You need to have a recent version of Node; we test using v6.9.1. It's possible you might already have Node installed, so to check what version you have, run:

   ```
   $ node --version
   ```

   If you have an older version of Node.js, or a version from the 11.x, you will need to download and install a version from the "stable" 10.x series.

   Once you've installed node, you need to make sure you have a current version of npm. Batavia requires npm v6.0 or greater; you can determine what version of npm you have by running:

   ```
   $ npm --version
   ```

   If you have an older version of npm, you can upgrade by running:

   ```
   $ npm install -g npm
   ```

   Once you've got npm, you can use it to install Batavia's JavaScript dependencies:

   ```
   $ npm install
   ```

5. Lastly, compile the Batavia library and bundle its dependencies:

   ```
   $ npm run build
   ```

Your final setup should end up looking like this:

```
_ beeware
  \_ batavia
  \_ venv (if using virtualenv)
```

You now have a working Batavia environment!

Next Steps

Next, we can *setup the sandbox*, and try out running Python in your browser. Or your can try running some Python code *from the command line*.

Troubleshooting Tips

After running "npm run build", if you receive the error:

```
"Module not found: Error: Cannot resolve 'file' or 'directory' ./stdlib"
```

Run this command:

```
$ python compile_stdlib.py
```

Then try compiling the Batavia library again:

```
$ npm run build
```

2.1.2 Tutorial: Deploying a Hello World application in the Batavia Sandbox

In this tutorial, you'll get the Batavia sandbox running, and use it to run a really simple "Hello, world!" program written in Python.

Prerequisites

This tutorial assumes you've read and followed the instructions in *the previous tutorial*. If you've done this, you should have:

- A `beeware` directory with a Batavia checkout,
- An activated Python 3.5 virtual environment, and
- Batavia installed in that virtual environment

Starting the test server

1. To start the test server, you'll need to be in the `testserver` directory under the batavia directory:

   ```
   $ cd testserver
   ```

2. Once you're in the `testserver` directory, you can install the requirements for the test server:

   ```
   $ pip install -r requirements.txt
   ```

3. Then you can start the test server

- On Linux/macOS:

  ```
  $ ./manage.py runserver
  ```

- On Windows:

  ```
  > python manage.py runserver
  ```

4. Now open your browser and go to http://localhost:8000

You should now see a page titled "Batavia testbed" divided into three sections:

- a top section with text box for entering your Python code and a "Run your code!" button
- a middle section with buttons for running sample Python scripts

- a *Console output* section at the bottom of the page that displays the results from running either your own code or any of the samples.

It's alive!

At the top of the page is a text box. This is where you can enter your Python code. Type the following into this text box:

```
print("Hello World!")
```

Then click on the "Run your code!" button. The page will reload, and below the area on the page named "Console output", you'll see the output you'd expect from this Python program:

```
Hello World!
```

Congratulations! You've just run your first Python program under JavaScript using Batavia! Now you can get a little more adventurous and try a loop. Replace your existing code in the text box with the following:

```
for i in range(0, 10):
    print("Hello World %d!" % i)
```

Click "Run your code!" again, and you should see the following on the screen in the console output section:

```
Hello World 0!
Hello World 1!
Hello World 2!
Hello World 3!
Hello World 4!
Hello World 5!
Hello World 6!
Hello World 7!
Hello World 8!
Hello World 9!
```

What just happened?

What happened when you pressed the "Run your code!" button?

When you clicked "Run your code!", your browser submitted the content of the text box as a HTTP `POST` request to the test server. The test server took that content, and compiled it as if it were Python code. It didn't *run* the code – it just compiled it to bytecode. It created the `.pyc` file that would have been produced if you had written the same code into a `test.py` file and ran `python test.py`.

Having compiled the source code into bytecode form, it then encoded the contents of the `.pyc` file into base64, and inserted that base64 string into the returned HTML response. If you inspect the source code for the page, you should see a block in the document `<head>` that looks something like:

```
<script id="batavia-customcode" type="application/python-bytecode">
    7gwNCkIUE1cWAAAA4wAAAAAAAAAAAAAAAAIAAABAAAAAcw4AAABlAABkAACDAQABZAEAUykCegtI
    ZWxsbyBXb3JsZE4pAdoFcHJpbnSpAHICAAAAcgIAAAD6PC92YXIvZm9sZGVycy85cC9uenY0MGxf
    OTc0ZGRocDFoZnJjY2JwdzgwMDAwZ24vVC90bXB4amMzZXJyddoIPG1vZHVsZT4BAAAAcwAAAA=
</script>
```

That string is the base64 encoded version of the bytecode for the Python program you submitted. The browser then takes this base64 string, decodes it back into a bytestring, and runs it through Batavia – a JavaScript module that does the same thing in a browser that CPython does on the desktop: interprets Python bytecode as a running program.

Push the button...

You may also have noticed a set of buttons between the text box at the top of the page and the Console output section. These are some pre-canned example code, ready for testing. Try clicking the "Run sample.py" button. Your browser should pop up a new window and load the BeeWare website. If you close that window and go back to the Batavia testbed, you should see a lot of output in the console section of the screen.

Inside the button

If you want to, you can inspect the source code. One part of `sample.py` that is of particular interest is the part that opens the new browser window:

```
import dom


print('Open a new web page...')
dom.window.open('http://beeware.org', '_blank')

print('Set the page title')
dom.document.title = 'Hello world'

print('Find an element on the page...')
div = dom.document.getElementById('pyconsole')

print('... and set of that element.')
div.innerHTML = div.innerHTML + '\n\nHello, World!\n\n'
```

What you should notice is that except for the `dom` prefix, this is the same API that you would use in JavaScript to open a new browser window, set the page title, and add some text to the end of an element. The entire browser DOM is exposed in this way, so anything you can do in JavaScript, you can do in Batavia.

You can even use this code in the sample code window: copy and paste this code into the "run code" text box, click "Run your code!", and you get a popup window.

Push the *other* button...

There are also a couple of "Run PyStone" buttons, each of which runs for a number of iterations. PyStone is a performance benchmark. On an average modern PC, the 5 loop version will be almost instantaneous; 500 loops will take less than a second; 50000 loops will take about 15 seconds. You can compare this with native performance by running the following in a Python (3.6 or earlier, as `test.pystone` was removed in 3.7) shell:

```
>>> from test import pystone
>>> pystone.main()
Pystone(1.2) time for 50000 passes = 0.521687
This machine benchmarks at 95842.9 pystones/second
```

You'll probably notice that Batavia is significantly slower than native CPython. This is to be expected – Batavia is going through a very complex process to run this code. It's not overly concerning, though, as the main use case here is basic DOM manipulation and responding to button clicks, not heavy computation.

As an aside, if you'd like to run more than a trivial performance benchmark, please check out pyperf and pyperformance.

2.1.3 Tutorial: Running Python code using Batavia from the command line

Batavia includes a simple command-line script that you can use to run your Python code. To use this tool you need to have followed the instructions from *Tutorial: Setting up your Environment*.

You can now run Python code from a code from the command line as follows:

```
npm run python /path/to/python_file.py
```

This runs the `run_in_batavia.js` script which in turn runs the Python code. This command will only work if you call it within the Batavia project directory and provide it the absolute path to the Python file to run.

You can alternatively directly run the `run_in_batavia.js` in Node. If you are not in the Batavia project directory you can still use this script as follows:

```
node /path/to/batavia/run_in_batavia.js /path/to/python_file.py
```

2.2 How-to guides

How-to guides are recipes that take the user through steps in key subjects. They are more advanced than tutorials and assume a lot more about what the user already knows than tutorials do, and unlike documents in the tutorial they can stand alone.

2.2.1 Setting up your development environment

The process of setting up a development environment is very similar to the *Tutorial: Setting up your Environment* process. The biggest difference is that instead of using the official BeeWare repository, you'll be using your own Github fork.

As with the getting started guide, these instructions will assume that you have Python 3 (currently supported versions are 3.5, 3.6, and 3.7).

Batavia codebase

Start by forking Batavia into your own Github repository; then check out your fork to your own computer into a development directory:

```
$ mkdir batavia-dev
$ cd batavia-dev
$ git clone https://github.com/<your github username>/batavia.git
```

Then create a virtual environment and install Batavia into it:

MacOS

```
$ python3 -m venv venv
$ . venv/bin/activate
(venv) $ cd batavia
(venv) $ pip install -e .
```

Linux

```
$ python3 -m venv venv
$ . venv/bin/activate
(venv) $ cd batavia
(venv) $ pip install -e .
```

Windows

```
> py -3.6 -m venv venv
> venv\Scripts\activate
(venv) > cd batavia
(venv) > pip install -e .
```

Install Node.js

Lastly, you'll need to install Node.js. You need to have a recent version of Node; we test using v10.x. Once you've installed node, you can use it to install Batavia's JavaScript dependencies, and compile the Batavia library:

```
(venv) $ npm install -g npm
(venv) $ npm install
(venv) $ npm run build
```

Running the test suite

You're now ready to run the test suite! This can be done the immediately-available-but-tiresome way, or the takes-little-effort-but-then-fun way, which is preferred (*assuming* your Python is configured for Tk; MacOS Python often is *not*).

For the fun way, you need to install BeeWare's test-running tool, Cricket. Follow the installation instructions to install it into your virtualenv; then, in the `batavia-dev/batavia` directory, type:

```
(venv) $ cricket-unittest
```

This launches a test-running GUI, where you can easily and intuitively run all tests or a subset of tests, see the progress of tests (which is quite valuable when running over 10000 tests), and whenever failure is encountered, immediately see the details.

If, for whatever reason, you want to run the tests without Cricket, you can always use a text test runner by typing:

```
(venv) $ python setup.py test
```

This will take at least several minutes, and can take upwards of 1.5hrs on most modern PCs/laptops. It will also generate around 10000 lines of console output - one line for each test that is executed. Each line will tell you the pass/fail status of each test - e.g.,:

```
test_abs_not_implemented (tests.builtins.test_abs.AbsTests) ... expected failure
test_bool (tests.builtins.test_abs.BuiltinAbsFunctionTests) ... ok
```

This indicates that tests have passed (`ok`), or have failed in an expected way (`expected failure`). These outcomes are what you expect to see.

If you see any tests reported as `FAIL`, `ERROR`, or `unexpected success`, then you've found a problem. If this happens, at the end of the test run, you'll also see a summary of the cause of those problems.

As soon as you see problems, you can stop the tests and start debugging. Cricket has a button for this; with the text test runner, hit Ctrl-C or Cmd-C to quit.

However, this *shouldn't* happen - Batavia runs continuous integration to make sure the test suite is always in a passing state. If you *do* get any failures, errors, or unexpected successes, please check out the *troubleshooting section* or get in touch, because you may have found a problem.

If you just want to run a single test, or a single group of tests with the text runner, you can provide command-line arguments.

To run a single test, provide the full dotted-path to the test:

```
$ python setup.py test -s tests.datatypes.test_str.BinaryStrOperationTests.test_add_
↪bool
```

To run a full test case, do the same, but stop at the test case name:

```
$ python setup.py test -s tests.datatypes.test_str.BinaryStrOperationTests
```

Or, to run all the Str datatype tests:

```
$ python setup.py test -s tests.datatypes.test_str
```

Or, to run all the datatypes tests:

```
$ python setup.py test -s tests.datatypes
```

Running the linter

```
$ npm run lint
```

2.2.2 Contributing to Batavia's code

In the following instructions, we're going to assume you're familiar with Github and making pull requests. We're also going to assume some entry level Python and JavaScript; if anything we describe here doesn't make sense, don't worry - we're more than happy to fill in the gaps. At this point, we don't know what you don't know!

This tutorial is also going to focus on code contributions. If your interests and skills are in documentation, *we have a separate contributors guide <:doc:'contribute-docs>'__* just for you.

Do the tutorial first!

Before you make your first contribution, take Batavia for a spin. The instructions in the *getting started guide <:doc:../tutorial/tutorial-0'>'__ should* be enough to get going. If you get stuck, that points to your first contribution - work out what instructions would have made you *not* get stuck, and contribute an update to the README.

Set up your development environment

Having run the tutorial, you need to *set up your environment for Batavia development <:doc:'development-env>'__*. The Batavia development environment is very similar to the tutorial environment, but you'll be using your own fork of Batavia's source code, rather than the official repository.

Your first contribution

In order to implement a full Python virtual machine, Batavia needs to implement all the eccentricities of Python behaviour. For example, Python allows you to multiply a string by an integer, resulting in a duplicated string (e.g., ‘‘ “foo”* 3‘‘ => "`foofoofoo`"). Javascript behavior can be quite different, depending on circumstances - so we need to provide a library that reproduces the desired Python behavior in Javascript.

This includes:

- all the basic operators for Python datatypes (e.g., add, multiply, etc)
- all the basic methods that can be invoked on Python datatypes (e.g., `list.sort()`
- all the pieces of the Python standard library that are written in C

As you might imagine, this means there's lots of work to be done! If you're looking for something to implement for your first contribution, here's a few places to look:

- Compare the list of methods implemented in Javascript with the list of methods that are available at the Python prompt. If there's a method missing, try adding that method.
- Look through the Javascript source code looking for `NotImplementedError`. Any method with an existing prototype where the Javascript implementation raises `NotImplementedError` indicates the method is either partially or completely unimplemented. Try to fill in the gap!
- Try writing some Python code and running it in Batavia. If the code doesn't run as you'd expect, work out why, and submit a pull request!

Getting Help

If you have any difficulties with this tutorial, or there's anything you don't understand, don't forget - we're here to help you. Get in touch and we'll help you out, whether it's giving a better explanation of what is required, helping to debug a difficult problem, or pointing you towards tutorials for background that you may require.

2.2.3 Contributing to Batavia's documentation

Here are some tips for working on this documentation. You're welcome to add more and help us out!

First of all, you should check the Restructured Text (reST) and Sphinx CheatSheet to learn how to write your .rst file.

To create a .rst file

Look at the structure and choose the best category to put your .rst file. Make sure that it is referenced in the index of the corresponding category, so it will show on in the documentation. If you have no idea how to do this, study the other index files for clues.

To build locally on GNU/Linux and open it on the browser:

Go to the documentation folder:

```
$ cd docs
```

Install Sphinx with the helpers and extensions we use:

```
$ pip install -r requirements_rtd.txt
```

Create the static files:

```
$ make html
```

Check for any errors and,if possible, fix them. The output of the file should be in the `_build/html` folder. Open the file you changed in the browser.

2.2.4 Implementing Python Built-ins in JavaScript

General Structure

JavaScript versions of Python built-in functions can be found inside the `batavia/builtins` directory in the Batavia code. Each built-in is placed inside its own file.

```
// Example: a function that accepts exactly one argument, and no keyword arguments

var <fn> = function(<args>, <kwargs>) {
    // These builtins are designed to be used only inside Batavia, as such they need␣
↪to ensure
    // they are being used in a compatible manner.

    // Batavia will only ever pass two arguments, args and kwargs. If more or fewer␣
↪arguments
    // are passed, then Batavia is being used in an incompatible way.
    // See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/
↪Functions/arguments
    if (arguments.length !== 2) {
        throw new builtins.BataviaError.$pyclass("Batavia calling convention not used.
↪");
    }

    // We are now checking if a kwargs object is passed. If it isn't kwargs will be␣
↪null. Like
    // obj.keys() in Python we can use Object.keys(obj) to get the keys of an object.␣
↪If the
    // function doesn't need support any kwargs we throw an error.
    if (kwargs && Object.keys(kwargs).length > 0) {
        throw new builtins.TypeError.$pyclass("<fn>() doesn't accept keyword␣
↪arguments.");
    }

    // Now we can check if the function has the supported number of arguments. In␣
↪this case a
    // single required argument.
    if (!args || args.length !== 1) {
        throw new builtins.TypeError.$pyclass("<fn>() expected exactly 1 argument ("␣
↪+ args.length + " given)");
    }

    // If the function only works with a specific object type, add a test
    var obj = args[0];
    if (!types.isinstance(obj, types.<type>)) {
        throw new builtins.TypeError.$pyclass(
            "<fn>() expects a <type> (" + type_name(obj) + " given)");
    }
```

(continues on next page)

(continued from previous page)

```
    // actual code goes here
    Javascript.Function.Stuff();
}

<fn>.__doc__ = 'docstring from Python 3.4 goes here, for documentation'

modules.export = <fn>
```

Adding Tests

The tests corresponding to Batavia implementations of built-ins are available inside `tests/builtins`. The Batavia test infrastructure includes a system to check the compatibility of JavaScript implementation of Python with the reference CPython implementation.

It does this by running a test in the Python interpreter, and then running the same code using Batavia in the Node.js JavaScript interpreter. It will compare the output in both cases to see if they match. Furthermore the test suite will automatically test the builtin against values of all data types to check if it gets the same response across both implementations.

In many cases these tests will not cover everything, so you can add your own. For an example look at the `test_bool.py` file in `tests/builtins`. You will see two classes with test cases, `BoolTests` and `BuiltinBoolFunctionTests`. Both derive from `TranspileTestCase` which handles running your code in both interpreters and comparing outputs.

Let's look at some test code that checks if a the Batavia implementation of `bool` can handle a bool-like class that implements `__bool__`.

```
def test_bool_like(self):
    self.assertCodeExecution("""
        class BoolLike:
            def __init__(self, val):
                self.val = val

            def __bool__(self):
                return self.val == 1
        print(bool(BoolLike(0)))
        print(bool(BoolLike(1)))
        """)
```

The `assertCodeExecution` method will run the code provided to it in both implementations. This code needs to generate some output so that the output can be compared, hence the need to print the values.

Process

For a given function, run *functionname.__doc__* in the Python 3.4 repl

Copy the docstring into the doc

Run the function in Python 3.4

Take a guess at the implementation structure based on the other functions.

Copy the style of the other implemented functions

2.2.5 Adding a module and testing it

Modules in Batavia can be implemented either natively, in JavaScript or in supported subset of Python.

To create a native module

Add the module path to `module.exports` in `batavia/modules.js`.

To create a module in Python

Add module to `compile_stdlib.py`.

To create a test

If the module doesn't exist yet, it must be created as a `test_NAME-OF-THE-MODULE.py` file.

If a test for the module already exists and you want to add functionalities to it, you must create a new function on the file.

To run the tests

On the Batavia directory:

```
$ python setup.py test -s tests.modules.NAME_OF_MODULE
```

For instance, to test the base_64 module:

```
$ python setup.py test -s tests.modules.test_base64
```

2.3 Reference

This is the technical reference for public APIs provided by Batavia.

2.3.1 JavaScript Compatibility Limitations

There are unavoidable high-level incompatibilities of some low-level Python fundamental functionality that do not exist in JavaScript. These are not temporary issues nor are they bugs that can be fixed, but limitations that result from the interaction between the two technologies.

These issues are detailed here.

id

"Return the identity of an object. This is guaranteed to be unique among simultaneously existing objects. (CPython uses the object's memory address.)"

This doesn't exist in JavaScript. There's no useful information that can be returned from *id* in this instance.

2.4 About Batavia

2.4.1 Frequently Asked Questions

Is Batavia a source code converter?

No. Batavia operates *at the bytecode level*, rather than the source code level. It takes the CPython bytecode format (the *.pyc* files generated at runtime by CPython when the code is imported for the first time), and runs that bytecode in a virtual machine that is implemented in JavaScript. No intermediate JavaScript source file is generated.

Isn't this the same thing as Brython/Skulpt/PyPy.js?

No. Brython and Skulpt are full implementations of the Python compiler and interpreter, written in JavaScript. They provide a REPL (an interactive prompt), and you run your Python code through Brython/Skuplt. Batavia doesn't contain the compilation tools - you ship pre-compiled bytecode to the browser, and that bytecode is executed in a browser-side virtual machine.

PyPy.js is a very similar idea, except that instead of being a clean implementation of the Python virtual machine, it uses Emscripten to compile PyPy source code into JavaScript (or the asm.js subset).

The biggest advantage of the Batavia approach is size. By only implementing the virtual machine, Batavia weighs in at around 400kB; this can be trimmed further by using tree-shaking to remove parts of Batavia that aren't used at runtime. This compares with 5MB for PyPy.js.

The easiest way to demonstrate the difference between Brython/Skulpt/PyPy.js and Batavia is to look at the *eval()* and *exec()* methods. In Brython et al, these are key to how the process works, because they're just hooks into the runtime process of parsing and evaluating Python code. In Batavia, these methods would be difficult to implement because Batavia compiles all the class files up front. To implement *eval()* and *exec()*, you'd need to run Batavia through Batavia, and then expose an API that could be used at runtime to generate new bytecode content.

How fast is Batavia?

Faster than a slow thing; slower than a fast thing :-)

Programming language performance is always nebulous to quantify. As a rough guide, it's about an order of magnitude slower than CPython on the same machine.

This means it probably isn't fast enough for an application that is CPU bound. However, if this is the case, you can always write your CPU bound parts in *pure* JavaScript, and call those directly from Python, same as you would for a CPython extension.

It should also be noted that Batavia is a very young project, and very little time has been spent on performance optimization. There are many obvious low hanging performance optimizations that could be explored as the project matures.

What can I use Batavia for?

The main use of Batavia is for writing web applications. Batavia provides the option for writing client-side logic in Python, rather than JavaScript.

What version of Python does Batavia require?

Batavia runs under Python 3.4, and compiles Python 3.4 compatible bytecode.

Why "Batavia"?

On 27 October, 1628, *Commandeur* Francisco Pelsaert took command of the *Batavia*, and with 340 passengers and crew, set sail from Texel. Their destination? The Spice Islands - or more specifically, island of Java in the Dutch East Indies (now part of Indonesia).

The Batavia was... a Java ship (rimshot!).

Interestingly, during the voyage, Ariaen Jacobsz and *onderkoopman* Jeronimus Cornelisz incited a mutiny, because they didn't want to go to Java - they wanted to escape to start a new life somewhere else. As a result of the mutiny, on 4 June 1629, the Batavia ran aground on Morning Reef, part of the Houtman Abrolhos, about 450km North of Perth, Western Australia, where this project was conceived.

The full story of the Batavia is known to most Western Australian schoolchildren, and is a harrowing tale of intrigue, savagery, and murder. It serves as a reminder of what can happen when you force people to go to Java :-)

The wreck of the Batavia was recovered in the 1970s, and now stands in the shipwrecks gallery of the Western Australian Maritime Museum.

2.4.2 The Batavia developer and user community

Batavia is part of the BeeWare suite. You can talk to the community through:

- @pybeeware on Twitter
- See the Getting Help page for information on where to ask questions regarding the use of the BeeWare suite, as well as links to discussions the development of new features in the BeeWare suite, and ideas for new tools for the suite.
- The original BeeWare Users and BeeWare Developers mailing lists, have been shut down, in favor of the getting help page mentioned above. However, the earlier messages and topics posted prior to its shutdown remain. See the announcement for more details on the shutdown.

Code of Conduct

The BeeWare community has a strict Code of Conduct. All users and developers are expected to adhere to this code.

If you have any concerns about this code of conduct, or you wish to report a violation of this code, please contact the project founder *Russell Keith-Magee*.

Contributing

If you experience problems with Batavia, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

2.4.3 Release History

0.1.0 - In development

Initial public release.

2.4.4 Batavia Roadmap

Batavia is a new project - we have lots of things that we'd like to do. If you'd like to contribute, providing a patch for one of these features:

- Port a set of basic type operations
- Implement a Python standard library module for JavaScript
- Implement StackMapFrames for the generated Java class files.
- Work out how to run the CPython test suite with Batavia